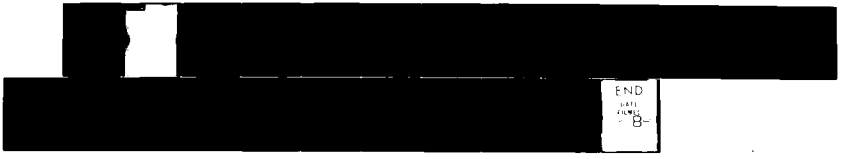
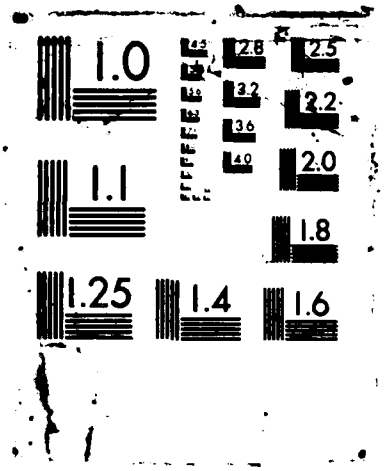
AD-A191 889 DEVELOPMENT ENVIRONMENT FOR SECURE SOFTWARE(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) C T SENNETT NOV 87 RSRE-87015 DRIC-BR-104812 1/1

UNCLASSIFIED F/G 12/5 NL



END
DATE FILMED
8-


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 87015

Title: The development environment for secure software

Author: C T Sennett

Date: November 1987

Summary

This report presents criteria to which a development environment should conform to be suitable for the production of secure software. It gives a rationale for the criteria and a suitable security policy model for software development.

THE DEVELOPMENT ENVIRONMENT FOR SECURE SOFTWARE

Introduction and motivation

The role of the development environment has not hitherto received a great deal of attention within the security community: the US Trusted Computer System Evaluation Criteria [DoD 1983] for example, clearly states the need for features within the development environment, but is not very specific about what they should be, apart from the requirement for generation and comparison tools. The controls exerted at the development stage are very important for security: as far as the threat to the operational system is concerned it is arguable that the greatest vulnerability occurs while it is under development. If the software can be attacked at this stage, particularly if it can be attacked after the completion of the evaluator's work, the fact of evaluation counts for nothing. It is also the case that software may be attacked most easily at this stage: the development tools are available; the attacker, if he is a member of the development team, will have intimate knowledge of the software and he may not be required to have the clearances needed to access the operational system.

The development environment must provide for protection during maintenance. After evaluation the system will change: even an evaluated system will have bugs in the untrusted software; it is likely that the operational requirement will evolve and the system may need to change to meet the needs of new hardware or new versions of an underlying operating system. Change and development of the system introduce a substantial vulnerability. Maintenance of the system will not be done by the original developers: typically, one may suppose that it is undertaken by service personnel, possibly on a short term of duty, or contract programmers, also frequently changing. Personnel on short terms of duty cannot be expected to appreciate the subtleties of the setting of ring numbers or privilege bits or other equally obscure machine dependencies which may be absolutely crucial to the security of the trusted computing base, so the development environment must protect against slips and errors arising as a result of lack of knowledge. It is unrealistic to expect the operational system not to have to undergo some changes during its lifetime, and it is equally unrealistic to expect the system to go through the complete evaluation process for every change. This report, therefore, addresses the major issue of what technical features are required of a development environment which would allow development of a system without re-evaluation.

A further motivation for a review of the role of the development environment in security arises from the need to integrate software verification systems, required for systems being developed to the highest levels of assurance. One may expect that, with the increasing industrialisation of the current verification systems, the requirement for a thorough going module system will increase, with independent development of modules of specification and proof, as well as independent development of the target software. The rigour of a formal system must depend on the consistency of the separate modules, which must surely be enforced by the development environment. Of equal importance is the question of keeping tight control on the transition from specification to the object code. There seems little point in formally verifiying conformance to a top level specification if access to the subsequent code is not tightly controlled. The facilities required to support these controls ought to be part of the specification of a secure development environment.

The development environment has a part to play within the process of evaluation itself. It must provide tools which support the role of the evaluator, but of more importance is the fact that it can provide controls which constrain the developer to produce an evaluatable system. If the development has taken place without thought for evaluation, the evaluation may be as costly as the original implementation. This will happen when the evaluation evidence is lacking, when documentation is not up to date, coding standards not enforced and when it is continually necessary to check that source text and object code agree. Practically, one would really like some assurance that the evaluation process will in fact terminate, preferably with a positive outcome,

and this will only be so if the development has been constrained to suit the evaluation. The development environment can support this goal by exerting controls, for example on the production of documentation and the carrying out of tests, to assure the evaluator that documentation and test plans exist for each item of software and that they are always up to date. With this sort of support the evaluation task becomes much better defined and quantifiable. It will also tend to reduce the effort spent in evaluation, but at the expense of effort spent in implementation: this however is desirable because review, testing, impact on design etc, are all better catered for within the development role, rather than within the evaluation role. Put more positively, it is more desirable to spend money on a better product than a better assessment of a product.

One final motivation for reviewing the role of the development environment is the current interest in project support environments for Ada. This is now coming to fruition in the form of specifications for preferred forms of APSE, for example CAIS [DoD 1986] and the similar European activity on a portable common tool interface, and it will be desirable to ensure that these accurately reflect the security requirements.

Control objectives for development environments

As has been indicated in the introduction, a major part of the security requirements within the development environment are concerned with various aspects of control, and it is necessary to specify the types of control and the objects over which the control is exercised. This is analogous to the control required for security in the operational system and I shall develop the view that a control policy, analogous to a security policy, should be specified for the development environment and that this should be expressed in terms of a control policy model, analogous to a security policy model. As with the security policy, criteria for acceptable implementations will need to be defined. The control policy is made up of three elements: an integrity policy, a security policy and requirements to support the role of the evaluator. Each of these will be dealt with in turn.

Integrity policy

The most important element in the control policy must be that which is concerned with the integrity of the trusted elements of the operational software. There are many vulnerabilities in this area. A prime concern must be to ensure that trusted code is not altered inadvertently, which is by no means a trivial matter as it is not practical to isolate the trusted software. In a large CCIS system for example, such software will occur at several points throughout the system: in workstations, network front end processors and database processors as well as in the central ADP system, so apart from occurring in many places it will have differing forms. For back-up and maintenance purposes this trusted code must be written repeatedly and assembled into consistent packages which gives many opportunities for the accidental omission of vital components. Note also that it is in the nature of security checks that a system still works if they are omitted.

An approach to this latter problem is to use special purpose system building tools which check for the presence of all the required portions of trusted code: the requirement then arises to ensure that the trusted system is only constructed with the trusted system building tool. A similar situation might arise if trusted code were required to be produced with a language subset: in this case one must ensure that the trusted code is only compiled with the subset compiler. Alternatively one might simply wish to ensure that potentially unreliable and untrustworthy software, obtained, say, from a bulletin board, was not used to develop a trusted system. In all these cases the requirement is to deny to untrusted tools the ability to create trusted software. In passing, one may note that the existence of trusted tools, ranging from trusted compilers to the development environment itself (which will also need maintenance) increases the amount of data in the system which must be protected from corruption.

Software under development may be deliberately subverted, a problem not confined to military systems: there are several published instances of fraud and of disgruntled

employees attempting to hold firms to ransom as a result of illicit software planted into trusted code. However, the peculiarly difficult problem presented by software implementing security mechanisms is that subversions of the mechanisms, almost by definition, do not make themselves manifest. A counter to the threat of deliberate alteration is to introduce individual accountability and the ability to constrain programmers so that they may only alter software they are responsible for, what one might call "need-to-alter".

Finally one may note the inclination that system developers have to produce insecure systems for testing purposes, so that particular conditions may be more easily set up and exercised. In principle these development systems present no threat, but they are easily confused with the operational system and may appear to behave in an identical way, apart from providing no security checks. It is important therefore to be able to separate experimental systems from operational systems.

Traditionally, these vulnerabilities have been addressed manually: one employs trusted programmers and a configuration management system to keep track of items of software making up the system to be delivered. This is not really an adequate approach. Most configuration management systems provide management tools which report the status of different versions and give the ability to assemble differing configurations, but do not exert adequate control. The problem is, supposing an old version of the system under development were to be assembled, what guarantees are there that the trusted parts are as they were when evaluated? What guarantees are there that when a new version is produced, it has the same security properties as the old? Typically, a new version is produced to cure a bug, probably not security specific. The cure for the bug may however alter some property of the code which is relied on by the system to guarantee its security, and this property could be quite obscure. It is almost impossible to maintain, throughout the life cycle of a project, a team of programmers who totally understand the consequences which alterations of code may have for the security of the system. In addition to this difficulty, it is not possible to introduce the concept of accountability without having the ability to audit operations which alter trusted software, and this must be done by the development system itself. Trusted software is only vulnerable within the development environment: it cannot be altered outside the computer, only destroyed, so manual controls on alteration are being exercised in an inappropriate environment. Quite apart from this, the scale of control required over many thousands of configurable items is entirely beyond human capabilities. Nor is it altogether believable that a manual system will not be circumvented from time to time as project deadlines draw nearer.

Consequently, controls must be exercised by the development environment itself. Nevertheless, the computer system must interact with the human system. It is important that listings of trusted code should be labelled as such, to act as a reminder to the programmers of the nature of the code they are writing. Trusted code should be distinguished from untrusted code and alterations to it should only be allowed to persons who are authorised to change it, and their actions should be accountable. These are all very familiar words and lead naturally to a requirement for a mandatory integrity policy, which is rather like a dual of the familiar mandatory security policy. This policy expresses the requirement that trusted objects should be marked as such and a user should have an integrity clearance which dominates the integrity classification of objects he is authorised to overwrite. Clearances may also be associated with tools to limit the ability of untrusted tools to operate on trusted software.

Integrity policies were first introduced by Biba [1977] and have been implemented within development environments (for example, see Sennett 1981), but are not widely used, largely because of difficulties associated with using them in practice. It is important therefore to choose the particular policy required very carefully. The most important property sought is that the integrity policy should be mandatory rather than discretionary. This supports the accountability objectives which can be related to the integrity clearances. The raising of an integrity classification should be an accountable action, involving a designated responsible officer, a two man rule or some other human guarantor. (The reduction of an integrity classification presents no

problems from the integrity point of view, although there are denial of service aspects which might make it desirable for it to be controlled in a similar way). Once an object has been classified to a given integrity level, a mandatory policy ensures that no user of the system can have any discretion to extend the ability to alter the object outside the set of users with the appropriate integrity clearance. Note however, that the requirement applies more to a given object rather than the information contained within it. As far as trusted code is concerned, the emphasis is on whether a particular object has been correctly handled rather than the nature of the information used to construct it. This is in contrast to security policies where control of information flow is very much of the essence of the problem.

The other aspect required in specifying a mandatory integrity policy is that it should be trustworthy. The assumption with any mandatory policy is that it is implemented as a central feature of the system, specified formally using a security or integrity model, with a high degree of assurance that the mechanism may not be by-passed.

Apart from the ability of the policy to confine users of the system into sets able to alter the code for given trusted elements, the introduction of integrity levels provides a basis for automatic labelling of source text and other appropriate handling rules. For this it seems desirable to decompose an integrity level into hierarchical levels and categories, in a similar fashion to security levels. First of all the hierarchical levels may be used to perform a gross discrimination on imported software (low, medium or high trustworthiness, for example), selective auditing (only alteration to high integrity data recorded), or invocation of special protective measures such as checksums (only for high integrity). The categories can be used to discriminate between projects, for example the use of project specific tools, control of access (for example, system updates only allowed from system consoles) as well as dividing the users into "need-to-alter" groups. Where data is common to two projects, it seems reasonable to require that tools and users should be cleared to both, so categories, rather than caveats (releases) are required.

The number of hierarchical levels will be determined by the number of different handling regimes within the development system itself. This is in distinction to security levels where there is a need to work with nationally defined levels, which may be related to handling rules outside the computer system and which are defined in terms of the damage caused by unauthorised disclosure. The difference arises because integrity rules are applied to data with a very limited circulation outside the computer (probably only magnetic tapes physically protected) and which is only vulnerable to corruption within it. The system and project specific emphasis means that the setting of integrity classifications and clearances may not necessarily be related to security rules. An integrity clearance should be granted only if the role of an individual requires him to alter trusted software and not on the basis of whether or not he has a security clearance; integrity clearances carry a specific responsibility and should be set on the basis of accountability. Integrity classifications should be determined by relating the protection afforded by the internal handling rules to the external vulnerabilities of the system, such as the trustworthiness and number of the people with access to it, as well as the damage caused by unauthorised alteration. It is possible, with a sufficiently small system confined to a small number of developers, to operate in "system high" integrity mode with no integrity policy at all.

A formal integrity and security model following an access control view of the world is given in the appendix and serves to give a precise definition of the access control policy proposed. Informally, the integrity policy proposed is as follows:

1. Integrity levels are made up of a hierarchical component and a set of categories with a dominance relation defining a partial order over the levels. A process is allowed to overwrite an object if the integrity clearance of the process dominates the integrity classification of the object.

2. The integrity clearance of the process is determined by a combination of the integrity clearance of the user instigating the operation and the combined

clearance of the current program or procedures being obeyed. An important policy point is that a program can only reduce an integrity clearance, so that the integrity clearance of a user sets an upper bound on his capabilities.

3. Data flow integrity is not observed: provided a program and a user have the clearance required to overwrite an object they are permitted to do so, regardless of the integrity of data which has previously been read. It is presumed that a program can ascertain what the integrity level of an input object is, if that is important.

Associated with this policy, but probably specific to a given system, would be requirements for identification and authentication appropriate to the physical and procedural environment and rules for selective audit and change of integrity label in the usual way. The lack of data flow integrity would probably require one to encapsulate certain tools, for example a standard compiler for the production of trusted code, with additional software to test the integrity of the input. Such encapsulation could be provided using a type mechanism discussed later.

The absence of data flow integrity is probably the most unusual feature of this policy and prevents it from being a strict dual of the security policy. The absence is intended to allow more flexibility in the system while maintaining a minimum level of mandatory control. With data flow integrity, experience shows that users tend to end up with data entirely at the top or bottom end of the integrity scale which defeats the purpose of introducing integrity levels. The combination of the simple integrity policy proposed here with the type mechanism and the ability to test the integrity of input data gives an adequate control for the purposes of producing secure software. Furthermore, it is not intended that this policy should be used to give gradually increasing levels of integrity to objects which have been processed by a sequence of assurance building tools, but rather that the controls provided should be used by a project manager to confine developers to the correct set of procedures for a given project, which may involve human intervention at various stages.

Security policy

A development environment will almost certainly be used to store classified data, but it is not quite so obvious that the software of the development environment should provide the access controls. The amount of classified data may be small, the developers may all be cleared to see it and the development will probably take place within a closed physical environment, as opposed to the operational system which could be extensively distributed. In theory therefore a system high mode of operation would normally be used, with controls provided by a discretionary security policy and individual accountability. However, the case is not quite so straightforward as this implies, so it is worth discussing the issue in the context of some typical scenarios as follows:

1. The development environment is used to produce software for a weapon system. The algorithms used are probably unclassified, but parameters for the algorithm and details of the application will almost certainly be classified.

2. The software developed for a typical CCIS system will also be unclassified, but the environment will need to contain classified supporting documentation (for example contractual specifications, configuration and deployment details).

3. Tools for security evaluation may be classified and encryption algorithms, including those used to protect passwords, may also be classified.

4. When the same environment is used for more than one project (the usual case) and in particular when the environment is used by a firm which is a member of a consortium, there will be strong need-to-know requirements to meet the needs of both military and commercial separation.

The last scenario makes it likely that some programmers with access to the environment will not have the clearance required for all the data stored on it, as well as increasing the requirement for need-to-know separation to such an extent that categories and a mandatory security policy become much more desirable. The aspect to note about the other scenarios is that the classified data is normally a small percentage of the whole, but it is stored in a system in which there is much copying, outputting and transport of data, on floppy discs, over networks and on magnetic tape. The possibilities for a hostile agent to hide classified data within large volumes of innocuous data are very large and operational procedures are unlikely to be able to cope. It seems therefore that labelling and a mandatory flow control policy will frequently be required and should be present in standard systems.

A feature of modern development environments is the existence of composite objects, such as Ada program libraries, and other more complex structures in which objects may be bound together by relations, as in a database. This raises the particularly important issue of the security policy requirements for the control over the establishing of relations between objects of differing classifications. The security treatment of these structured objects will depend on the degree of assurance required for the implementation of the security policy. A high assurance is not consistent with a fine granularity of protection: if high assurance is required for a given application it will probably be achieved independently of the detailed design of the development environment, for example by separating whole databases. A lower degree of assurance will permit the security policy to be extended to the atomic objects of the environment, which is certainly desirable, but has many consequences for the detailed form of the security policy. These issues are addressed in detail in the security model given in the appendix which proposes the minimum technical requirements.

Evaluation policy

These two elements of mandatory integrity and security policy together with standard discretionary access controls, go some, but not all of the way to meeting the control objectives in the development environment. There remain vulnerabilities not countered by these controls. A typical and most frequently occurring one arises from the fact that when an evaluator is assessing some code he will normally be looking at the source text, rather than the loadable object code itself. This raises questions as to whether the object code actually corresponds to the source text and whether it will be the code which is actually obeyed within the operational system. What is required is a control mechanism to ensure that source text and corresponding object code can be inextricably bound together and that operational code has well defined origins. The control requirements for consistency and identification which appear here are also needed at various other points within the evaluation process. Specification and design must be shown to be consistent; a verification condition must be related to the specification or software which gave rise to it; a proof can be invalidated by alterations in antecedents which in turn are derived from a specification which may have been altered. In all of these cases one is required to establish consistency between various representations of an underlying object. An interesting variation occurs when the implementor carries out part of the test programme for evaluation: the evaluator will need to be assured that the test has indeed taken place and that the correct input data was used and the anticipated results were obtained, otherwise the evaluator will need to repeat the test himself. In other words, the evaluator must be sure that the test results are consistent with, that is, derived from, a test of the software actually being evaluated.

The above examples indicate that control is required at many points in the production process. As far as the environment is concerned, this is by no means standardised, nor is it desirable that an environment should only support one concept for the software development process. At the very least it will have to support the production of both trusted and untrusted software and one may expect projects to use differing sets of tools according to project requirements and tool availability. The environment must therefore support some generic capability for control which may be invoked to a greater or lesser extent according to the set of tools in use and the degree to which the software is to be trusted. In each of the examples above, the essential

requirement is that if Bs are derived from As then it must be possible from a given B to find the A which gave rise to it. In some cases, a slightly easier requirement that Bs can only be derived from a certain set of As is acceptable, trusted code from evaluated code for example.

Put in this way, the generic requirement suggests the need for a typing system in which objects under the control of the development environment have a type and may only be constructed using the appropriate functions. Each operation provided by the environment takes as input objects of a certain type and also delivers objects of a certain type. The nature of the control which may be provided by this mechanism can be illustrated by the following fragment of a hypothetical specification for a development environment which deals with the facilities for evaluation and entering into configuration control. The specification language Z [Sufrin 1983, Hayes 1987] will be used, with the type system for this language being a model for the types required in the development environment.

Documentation and code are represented by given types which are otherwise uninterpreted in this fragment:

```
[doc, code]
```

One can construct a type for trusted software, depending, say, on whether it is for a central system, a workstation, or a network component:

```
trusted_sw ::= central«code» | ws«code» | network«code»
```

This Z datatype definition introduces a type, `trusted_sw`, and functions for constructing objects of this type, `central`, `ws` and `network`. The constructor functions have an actual implementation and the access control to the functions ensures that, say, the workstation team does not have access to the `central` function and consequently cannot affect central trusted code, even though they have the integrity clearance required to produce trusted code for their project. Configurable software consists of either trusted or untrusted items: suppose that trusted items must be accompanied by evaluation evidence, in this case taken to be documentation, not required of the untrusted software:

```
configurable_code ::= trusted«(doc × trusted_sw)»
                    | untrusted«code»
```

The `untrusted` constructor can be made freely available, allowing anyone to produce configurable items from untrusted software but only the evaluator is provided with the `trusted` function which constructs configurable items from trusted software, thus ensuring that the teams producing trusted software can only get their software entered into configuration control as a result of evaluation. The functions implementing the constructors can also check that trusted code has the appropriate integrity classification thus giving a guarantee that whatever version of a system the configuration management system assembles, its trusted component has been protected from unauthorised modification.

Although this is only a toy example, it does illustrate the power of the type checking mechanism. For this to form a satisfactory control mechanism there must be three elements present:

1. The ability of the user to define types (such as `trusted_sw` and `configurable_code` above) together with their associated constructors (such as `trusted` and `untrusted` for `configurable_code`).

2. The assignment of types to objects should be under system control and types should be unforgeable.

3.

The ability to obey the constructor functions and more complex functions built up from them must be subject to discretionary and mandatory access control.

Given these elements in the example, and an appropriate distribution of the access control rights, one can ensure that trusted code is the separate responsibility of a central, workstation or network team; that trusted software is not entered into configuration control without being evaluated and that it cannot be evaluated without providing the evaluation evidence.

This example can be extended and modified to cover other management structures and other control attributes: for example, the binding of source text and object code can be addressed by defining a type for modules. Module values may be made up using a constructor which takes both source text and compiled code as arguments, thus binding the two together. If the only way to make a module is via a trusted function which binds its source text with the compiled code in this way, the evaluator does not have to be concerned with consistency between the two: the fact that the object he is handling is a module guarantees this.

Experience with computer languages indicates that type checking can be carried out efficiently and with a high degree of assurance: it is flexible and seems to be ideally suited to carry out the controls required in a development environment.

Assurance

These three aspects of integrity, security and type checking together constitute the control policy of the development environment. The question now arises as to the assurance required for their implementation. For example, if the development environment is to produce formally verified software, should this control policy itself be formally verified? The topic is not quite so straightforward as might appear. One may argue that the amount of data controlled by the development environment may well be much less than that in the operational system, and so the assurance requirements are less: conversely one may argue that a fault in the development environment may allow illicit access to all of the data in the operational system and so the assurance requirements should be more. Again, one may argue that the users of the development environment are fewer than for the operational system, but this is countered by the fact that they have access to the full facilities of the development system.

It is clearly rather difficult to give an overall assurance level because the situation of each project may be quite different, with different integrity and denial of service vulnerabilities; moreover, there are no standards at the moment for integrity requirements. One may note also that the three components of policy within the development environment are supported to differing degrees within current computer architectures. Much experience has been gained in the use of existing machines in support of standard security policies, and this experience should also apply to integrity policies, but type checking to the formally verified levels of assurance is best supported by more advanced architectures, ideally capability based, [see, for example, Karger and Herbert 1984, Wiseman 1986] so it may be unrealistic to expect this level of assurance in the implementation of type checking in the near future.

As far as requirements documents such as the RAC [DoD 1986] are concerned it should only be necessary to specify functionality, leaving the possibility of implementations at differing assurance levels. However, it does seem desirable to indicate the likely level of assurance which would meet most needs at a reasonable cost. Current practice is to develop software under a system high mode of operation, for which the minimum security requirements would have relatively modest criteria for assurance of the correctness of such controls as were imposed. It has been argued above that this is not adequate and that security, integrity and type checking policies should be present; however, it is likely that most uses of the development environment would not require extreme levels of assurance. Furthermore, a granularity of protection at the level of, say, a file of source text corresponding to one configurable item would be

adequate, and covert channels and inferences on database information are unlikely to be a problem.

This gives a framework for a minimum acceptable implementation of a development environment to meet the requirement. In future, these standards are likely to rise: there are real vulnerabilities to be countered, and the whole point of IPSE development has been to provide an environment for the development of software in a more controlled fashion than it is at present. IPSE developments should therefore follow their own rationale and be produced to high standards and reliably. The ultimate pressure for trustworthy IPSEs will undoubtedly come from the requirement to reduce the costs of re-evaluation to a minimum.

Security requirements for modules and object code

So far, requirements for access control have been discussed, which may be regarded as standard functionality which a development environment, or indeed any operating system, should provide for its tools. Further security requirements are concerned with the interface between tools, in particular with the handling of object code in the system. Security requirements here are particularly necessary when the implementation is required to be verified. For these levels of trust it is highly desirable to have a formally defined representation of the code, to be used as the common compiler target language. As this approach brings many benefits to security at all levels of trust, and is in fact generally useful, it is desirable that a formally defined representation of object code should be part of the security requirements. Work at RSRE has for some time been based on defining the properties of such a language [Core and Foster 1986] and this has established the feasibility of the concept.

Verification of implementation is concerned with the verification of properties of software at some level at or below the level of a language such as Ada. Current verification systems do not tackle languages as complex as Ada as this results in an unmanageable task, so the approach is generally to use some simple (although still high level) language, whose properties can be formally specified and which is consequently a suitable object for formal verification. This language can be translated into a standard language such as Ada or Pascal and the verified code incorporated into the total system. This approach brings with it a large discontinuity in the chain of assurance: having gone to the trouble of writing in an especially simple language for the purposes of verification, what guarantees can there be that the more complex standard language does not violate the proven security properties, particularly when combined with other items of software. Writing a compiler for the verification language does improve the situation, but one still has the possibility that unless the hardware protection can be exploited, the generated code may still be incompatible, as far as the verified properties are concerned, with the code generated for other parts of the system.

A formally specified common intermediate language can address this problem by precisely defining an abstract target machine and its interactions, independently of the quirks of particular hardware. This would allow trusted and untrusted code to be combined together with some assurance that they were not interfering with each other. Thus the verification system should produce intermediate language output. The properties of this code may be investigated directly, or one may trust the compiler, both approaches being supported by the formal definition of the intermediate language. The verified intermediate language may now be safely combined with other code which has not been verified simply by virtue of the consistency properties of the language itself. The RSRE development already referred to supports this consistency check using a type system which may be extended into the type system of the development environment itself and integrated with the other type controls already specified. Assurance of the correct checks being made will depend upon the translator from the abstract machine to the real machine and given that the abstract machine does not have to have features supporting human readability there seems to be no reason to suppose it cannot be implemented to the requisite standards; certainly

one could have more assurance in such a translator than a compiler which went directly from a high level language to machine code.

A sufficiently general intermediate language can also address the problem of interactions and inconsistencies within a formal verification system. In such a system, the code of a module will correspond to the abstract syntax of the specification language or the logical calculus being used. This abstract syntax is a data structure which has a machine representation like any other data structure and there seems no reason not to use the formally defined target language and its module mechanism to represent it. Of course a module of specification may not make much sense to a module of Ada, but then this is just an extreme example of the language incompatibilities which also make it difficult for an Ada module to call a Pascal module. The benefits of using a common mechanism are that it centralises a vital part of the mechanism which maintains the integrity of the system; and it will of course also benefit the users to have a common interface.

Once the principle of a common intermediate language has been accepted, a number of benefits accrue. The first is simply the advantage of host target working. Most large scale systems are distributed and made up of a number of components, so the development environment will need to be able to produce code for a number of differing target machines. Quite apart from the operational system, the development environment itself will be a distributed system, probably having different machines for the implementors and the evaluators. It is helpful to have a common form for all code so that it may be evaluated and treated uniformly and easily transported to its destination. The disparity between implementation system and operational systems is particularly great when the latter are highly trusted: these systems tend to be small, in contrast with the advanced and large scale workstations used for their formal verification. And again it is worth emphasising that, because of the portability and formal definition, one may have confidence that what has been proved about an implementation on the workstation used for verification will also hold on the operational machine.

Portability and formal definition also support re-usability. Again, experience of verified systems tends to indicate that because the production process with verification is so long, the underlying hardware is obsolescent when introduced into service; this experience of course is not peculiar to software verification, but it would certainly be highly desirable to be able to re-use software with certain guaranteed security properties, simply from the point of view of the investment which has been made in it.

The standard intermediate language supports the portability of tools in general, but of particular interest to security are those concerned with testing. Secure software must, of course, be extensively tested and it almost goes without saying that a test harness will, as a result, be required. Not only is it the case that the number of tests is large and they have to be repeated for re-evaluation, but also tiger-team testing must involve taking the software through unusual regions of its state space, exploring the effects of errors and exceptional conditions and so on. It is almost impossible to do this without a test harness which can simulate these exceptional environmental events. The benefit which a common target language is able to give here is that it is only necessary to write one test harness, rather than one for each new project. The portability of the target also means that one can carry out tests on the evaluator's machine and have confidence that they represent reality on the target machine.

Another set of tools of interest to security are those concerned with analysis. Analysis of software is an indispensible element in evaluation: it may be used at simple levels, to check the quality of the evaluated software or the completeness of testing, through to complex levels to check the detailed logical structure of a program. Again, a common target language requires only one set of analysis tools with corresponding economies. In this respect, our experience at RSRE has led us to value particularly the algebraic approaches to the specification of target languages. Given an algebraic description of the target machine, particular properties under analysis can be

investigated by defining homomorphisms on the algebra, an approach which lends itself to the rapid development of a whole range of analysis tools.

Conclusion

At the present moment standards are being set for development environments which will be with us for the next 10 or 20 years. The facilities within the development constrain both the achievable level of security and the cost of maintaining it in operational systems. It would be most unfortunate if such environments could not be used for the production of secure systems, but equally unfortunate if their use added substantially to the burden and costs of producing secure software. The facilities proposed here add substantially to the assurance one can have in the secure operation of operational systems, but for the most part they are not peculiar to security. Integrity features are required for any high assurance software and will be necessary for many civilian as well as Defence requirements, so the security requirements proposed here are not additional to those which are, or will be, required in the civilian sector.

REFERENCES

Biba, K J. "Integrity considerations for secure computer systems," ESD-TR-76-372, ESD/AFSC Hanscom AFB, Bedford, Mass., USA, April 1977 (MITRE MTR 3153, NTIS AD A039324)

Department of Defense. "Trusted Computer System Evaluation Criteria", CSC-STD-001-83, National Computer Security Center, Fort Meade, Maryland, USA, August 1983.

Department of Defense. "Requirements and design criteria for the common APSE interface set", Report prepared for the Ada Joint Program Office, the Pentagon, Washington, DC 20301-3081, USA, October 1986.

Core, P W and Foster J M. "Ten15: an overview", RSRE Memorandum 3977, 1986

Hayes I. (ed) "Specification Case Studies", Prentice Hall International series in Computer Science, 1987.

Karger, P A, Herbert, A J. "An augmented capability architecture to support lattice security and traceability of access", Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, USA, April 1984.

Sennett, C T. "A security package for the George 3 operating system", RSRE Report 81017, November 1981.

Sufrin, B. "Formal system specification - notation and examples", in "Tools and Notations for Program Construction" (Neel ed.), Cambridge University Press, 1983.

Wiseman, S. "A secure capability computer system", Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, USA, April 1986.

APPENDIX - A COMBINED SECURITY AND INTEGRITY POLICY MODEL

This appendix is devoted to a Z specification of a combined security and integrity policy model. The aim of the specification is to capture the access control rules precisely while leaving most of the semantics and computational model unspecified; however, in view of the integrity policy required, it is necessary to model some aspects concerned with program execution, which do not appear in a standard Bell and LaPadula description.

First of all, security and integrity levels, dominance relations ($\leq$) and greatest lower bound (glb) and least upper bound (lub) operators will be assumed, which together form a lattice:

$$[Sy, Iy]$$

$$\_\leq_S\_ : Sy \leftrightarrow Sy$$
$$\_\leq_I\_ : Iy \leftrightarrow Iy$$

$$(\_\leq_S\_) \in partial\_order\ Sy$$
$$(\_\leq_I\_) \in partial\_order\ Iy$$

$$glb_S : (Sy \times Sy) \rightarrow Sy$$
$$glb_I : (Iy \times Iy) \rightarrow Iy$$
$$lub_S : (Sy \times Sy) \rightarrow Sy$$

$$\forall r, s, t : Sy \bullet (r\ glb_S\ s) \leq_S r \wedge (r\ glb_S\ s) \leq_S s \wedge t \leq_S r \wedge t \leq_S s \Rightarrow t \leq_S (r\ glb_S\ s)$$

$$\forall i, j, k : Iy \bullet (i\ glb_I\ j) \leq_I i \wedge (i\ glb_I\ j) \leq_I j \wedge k \leq_I j \wedge k \leq_I j \Rightarrow k \leq_I (i\ glb_I\ j)$$

$$\forall r, s, t : Sy \bullet r \leq_S (r\ lub_S\ s) \wedge s \leq_S (r\ lub_S\ s) \wedge r \leq_S t \wedge s \leq_S t \Rightarrow (r\ lub_S\ s) \leq_S t$$

The operations to be controlled are introduced as a given set (OP) and are not further specified apart from being categorised as a read, write, overlay (read and write) or an execute operation, as given in the definitions below. Other operations of the computer are assumed to cause no flow of information among the objects of the system and are not dealt with in the model.

$$[OP]$$

$$read, write, overlay, execute : \mathbb{P}\ OP$$

$$(read, write, overlay, execute)\ partition\ OP$$
$$\bigcup \{read, write, overlay, execute\} = OP$$

Values in the machine are also treated as a given set and are defined as being either data, program or composite in the sense of being constructed from other values. The more general case of related objects is discussed at the end of this appendix. Data is not further interpreted within the model, but program is considered to be a sequence of operations and names: the names are looked up in an environment to provide the object which the operation will act upon. Composite objects are needed in a security model for development environments in order to cater for program libraries: the fact

that one module in a library is classified should not require programs which do not use that module to be classified also. Composite objects are modelled as sets of simpler objects, while objects themselves have a content which is a value together with security and integrity classifications:

$$[Name, Value]$$

Object

$$content : Value; \quad class : Sy; \quad iclass : Iy$$

$$data : \mathbb{F}\ Value$$
$$program : seq(OP \times Name) \rightarrowtail Value$$
$$composite : \mathbb{F}\ Object \rightarrowtail Value$$

$$(data, rng\ program, rng\ composite)\ partition\ Value$$
$$\bigcup \{data, rng\ program, rng\ composite\} = Value$$

Each operation in the model is assumed to occur on behalf of a user and this security model prescribes what sequences of operations are allowed to occur for a given user. The usual Bell and LaPadula-like rules on security will be followed by maintaining the range of permitted security levels for reading and writing, while the integrity rules are defined using a current integrity level. These are gathered together in the schema below.

Levels

$$hiso, losi : Sy; \quad hisi : Iy$$

A read access is allowed for any object classified up to hiso (highest source); a write access is allowed for any object whose classification is at least losi (lowest sink) and integrity classification no greater than hisi (highest sink). Read and write operations change losi and hiso respectively according to the schemas given below, as a result of which losi always dominates the classification of any data which the process has read access to and hiso is dominated by the classification of any object the process can overwrite, which thus expresses the usual security rules. Note that this model is a specification and the implementation is not constrained necessarily to have variables to contain these levels.

Operations alter the state associated with a given user; as far as the model is concerned most state changes are irrelevant as it is only concerned with security and integrity aspects. For these purposes the state must contain the security levels, as above and the environment, to give a meaning to program. The convention of having a distinguished object to represent the parameter of the current operation will also be adopted. This will be the source or destination of the data when the operation is a read or write operation respectively and is included within the state space associated with the user.

State

$$Levels$$
$$Object$$
$$env : Name \nrightarrow Object$$

A read operation is any operation which may result in data flowing from the object into the user's data space, apart from the state space modelled here; while a write operation may result in data flowing in the opposite direction. The model is based on an access control view of the world in which the accesses permit subsequent

movement of data, after the manner of opening a file. It is assumed that the environment is only accessible as a result of obeying program in the manner constrained by the model, which preserves security. Consequently the presence of a high security object in the environment presents no threat; however, objects added to the environment by an operation must have at least the classification of data currently contained in the user's state space, which is no more than losi. This constraint is applied to all state changes (including those for operations not described by OP above) and gives the security rule for the creation of objects. This expresses the fact that if an operation adds one or more objects to the environment, the classification of the object must be at least losi'.

ΔState

State; State'

∀ obj : Object | obj ∈ rng(env'\env) • losi' $\leq_S$ obj.class

An initial value for the state can be set at any time when the user's state space is guaranteed free from classified data, normally at login or the commencement of a transaction. Only the security levels are significant in this case, when hiso should be set to the users clearance, losi to the bottom element of the security lattice and hisi to the user's integrity clearance, assuming the program currently executing is at the top integrity level.

The effect of the operations is governed by a function

execute : (State × OP)→State

which determines what state changes are brought about by a given operation. The model consists of a partial specification for execute in the form of constraints for each sort of operation and which governs the effect of sequences of operations starting from a given state. The model only constrains security levels so the object in the new state and the values in the new environment are left unspecified. The specification for execute will be built up incrementally according to the type of operation and the nature of the object it is applied to.

First of all, the read operation has a different effect depending on whether the current object is composite or not. For read operations on simple objects, the effect is null if the current working clearance does not allow it, but if it does, losi is changed to constrain future write operations:

Sreadops

ΔState
op : OP

op ∈ read
content ∈ data
class $\leq_S$ hiso ⇒ losi' = losi lub_S class
hiso' = hiso ∧ hisi' = hisi
¬(class $\leq_S$ hiso) ⇒ θState = θState'

The effect of a read operation on a composite object is simply to introduce some or all of the constituent objects into the user's environment, without any change in the working clearances. For this to be an allowable operation, the constituent objects

must be protected at least as well as any other object, which may be difficult to achieve for high levels of assurance.

Creadops

$$State; State'$$
$$op : OP$$

$$op \in read$$
$$content \in rng\ composite$$
$$rng(env' \backslash env) \subseteq composite^{-1}\ content$$
$$\theta Levels = \theta Levels'$$

Write operations either to a simple object or a composite one are permitted if the object has a classification at least that of any object read. However, the integrity constraint must also be taken into account:

writeops

$$\Delta State$$
$$op : OP$$

$$op \in write$$
$$content \in data \vee content \in rng\ composite$$
$$losi \leq_S class \wedge iclass \leq_I hisi \Rightarrow hiso' = hiso\ glb_S\ class$$
$$losi' = losi \wedge hisi' = hisi$$
$$\neg(losi \leq_S class \wedge iclass \leq_I hisi) \Rightarrow \theta State = \theta State'$$

In this simple model, executable objects may be created or deleted, but not overwritten. In principle, there is no reason to forbid the overwriting of program, although it may be necessary to distinguish the integrity classification for overwriting the object from its clearance when being obeyed (see below).

After a write operation, subsequent read operations are constrained to a security level no greater than that of the object just written. This follows from the access control view of the world in which it is assumed that once an object has been accessed for writing, data may flow into it at any time afterwards. If this is not the case, $hiso'$ can be left at the same value as $hiso$. Overlay operations are simply a combination of the two:

Soverlayops

$$\Delta State$$
$$op : OP$$

$$op \in rng\ overlay$$
$$content \in data$$
$$class \leq_S hiso \wedge losi \leq_S class \wedge iclass \leq_I hisi \Rightarrow$$
$$hiso' = hiso\ glb_S\ class$$
$$losi' = losi\ lub_S\ class$$
$$hisi' = hisi$$
$$\neg(class \leq_S hiso \wedge losi \leq_S class \wedge iclass \leq_I hisi) \Rightarrow$$
$$\theta State = \theta State'$$

Coverlayops

$$\Delta State$$
$$op : OP$$

$$op \in rng\ overlay$$
$$content \in rng\ composite$$
$$losi \leq_S class \wedge iclass \leq_I hisi \Rightarrow$$
$$hiso' = hiso\ glb_S\ class$$
$$rng(env' \backslash env) = rng\ composite^{-1}\ content$$
$$hisi' = hisi$$
$$\neg(losi \leq_S class \wedge iclass \leq_I hisi) \Rightarrow \theta State = \theta State'$$

Execute operations must be applied to program, and for this it is necessary to have a function for the execution of sequences of operations:

$$executeseq : (State \times seq\ (OP \times Name)) \nrightarrow State$$

$$\forall s : State;\ ops : seq\ (OP \times Name) \bullet$$
$$\#ops = 1 \Rightarrow executeseq(s, ops) = s''$$
$$\#ops > 1 \Rightarrow executeseq(s, ops) = executeseq(s'', tl\ ops)$$
$$where$$
$$s' \triangleq \mu State \mid env = s.env \wedge \theta Object = env\ (snd\ hd\ ops) \wedge hiso = s.hiso \wedge losi = s.losi \wedge hisi = s.hisi \bullet \theta State$$
$$s'' \triangleq execute(s', fst\ hd\ ops)$$

This function represents the model of execution of a program. It is defined by induction on the sequence: the first operation in the sequence is applied to the object found in the current environment from the first name in the sequence; subsequent operations take place within the state changed by the first operations. Exceptions have not been modelled for reasons of clarity, but the execution model may be easily extended to cater for this by allowing premature termination of the sequence. This function can now be used to define the constraints applied to execute operations:

executeops

$\Delta State$
$op : OP$

$op \in execute$
$content \in rng\ program$
$hisi' = hisi$
$hiso' = hiso_2 \wedge losi' = losi_2$
$env' = env_2 \wedge \theta Object' = \theta Object_2$
where
$State_1;\ State_2$

$hisi_1 = hisi\ glb_I\ iclass$
$hiso_1 = hiso \wedge losi_1 = losi \wedge env_1 = env$
$\theta Object_1 = \theta Object$
$\theta State_2 = executeseq(\theta State_1, program^{-1} content)$

In this schema, $State_1$ is the state for the start of the execution of the object, in which $hisi$ is modified if necessary to be no greater than the clearance of the program, represented by $iclass$. In this model, any program may be executed, the control of access to the program itself being provided by discretionary security. This follows from the view that programs may only diminish clearances, so there is no point in having a mandatory access check to the program itself. $State_2$ is the state after execution of the program which is used to derive the final state after the completion of the execute operation in which $hisi$ is restored to its initial value.

Finally the separate constraints may be combined together to give a partial specification for the security properties of execute.

$$ops \triangleq Sreadops \vee Creadops \vee writeops \vee Soverlayops \vee Coverlayops \vee executeops$$

$$\forall ops \bullet \theta State' = execute(\theta State, op)$$

This covers the case of a simple system in which the structure of the system is given only by the environments and the structure within the composite objects. More advanced systems will allow the setting up references from one object to another to form a general lattice structure rather than a hierarchy. This may be modelled by extending the Object schema and introducing a reference value to replace composite objects.

$data : \mathbb{F}\ Value$
$program : seq(OP \times Name) \rightarrowtail Value$
$reference : Object \rightarrowtail Value$

$(data, rng\ program, rng\ reference)\ partition\ Value$
$\cup \{data, rng\ program, rng\ reference\} = Value$

Relations between objects are many and various within development environments; however, the security properties may be discussed in terms of a scheme in which relations are identified by a key (represented by a given set); they are associated

with an attribute (modelled as unstructured data) and they point to an object (represented by a reference). Different schemes are possible, including many to many relationships, but this contains all the security relevant possibilities.

[key]

Object

content : Value; class : Sy; iclass : Iy
relations : key ⇸ (data × rng reference)

The basic assumptions which it seems reasonable to make about the security policy are that, for development environments, knowledge of the existence of a classified object is no threat to security and that a relation between objects of differing integrity levels is no threat to integrity. This means that, a priori, there is no reason why objects having differing security levels should not be linked, and that the simple act of navigating through an object's relations need require no security checking. The reason for this is that the reading an object found as a result of the navigation will be checked using the usual rules given above. There will, however, be covert channels associated with creating and deleting objects which will be discussed later. First of all, it is necessary to discuss the addition and deletion of relations. This will be done within the context of the following schema which specifies the addition of a relation.

Add_relation

ΔObject
k : key; d : data; r : rng reference

---

content' = content ∧ class' = class ∧ iclass' = iclass
relations' = relations ⊕ {k ↦ (d, r)}

The first thing to note is that the attribute is not treated as a separate object with its own classification, consequently the operation of adding a relation must be treated as a write operation on the object and subject to the rules of writeops given above. However, because the operation of adding a relation is a discrete operation, it will not be necessary to change hiso. The reading of an attribute is a read operation on the object, and so must be subject to the rules of Sreadops. No check is required on the classification of the object pointed at by the reference and this, together with no check being required on navigation is equivalent to the constraints of Creadops. If it is necessary to form a bi-directional link between two objects of differing security levels, or to form a cycle where the objects are not at the same level, these security rules require the relation to be added when losi is dominated by the security level of the lowest object, so the attributes added to the relations are constrained to have a classification no greater than the lowest. Note that writeops allows for writing an object at a higher classification than losi. Deletion of a relation is also a write operation, so again in the case of bi-directional relations, it must be carried out at the lower of the two classifications.

Somewhat surprisingly, these security rules are not particularly constraining, but do require the implementation to guarantee that the ability to add a relation does not in any way allow the transfer of information from the contents of the object referenced, which is hard to achieve at the highest assurance levels. They carry with them a covert channel, arising from the ability to navigate through a high security object. If such an object is accessible to processes at differing security levels, it is possible for the high security process to signal to the low security process by adding and deleting relations. This is a relatively high bandwidth channel, but it is also fairly easily monitored: the addition and deletion of relations is an infrequent operation, so unusual behaviour is relatively easily detected.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ...UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Report 87015	3. Agency Reference	4. Report Security Classification UNCLASSIFIED
5. Originator's Code (if known) 778400	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment St Andrews Road, Malvern, Worcestershire WR14 3PS		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

DEVELOPMENT ENVIRONMENT FOR SECURE SOFTWARE

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Sennett C T			1987.11	19

11. Contract Number	12. Period	13. Project	14. Other Reference

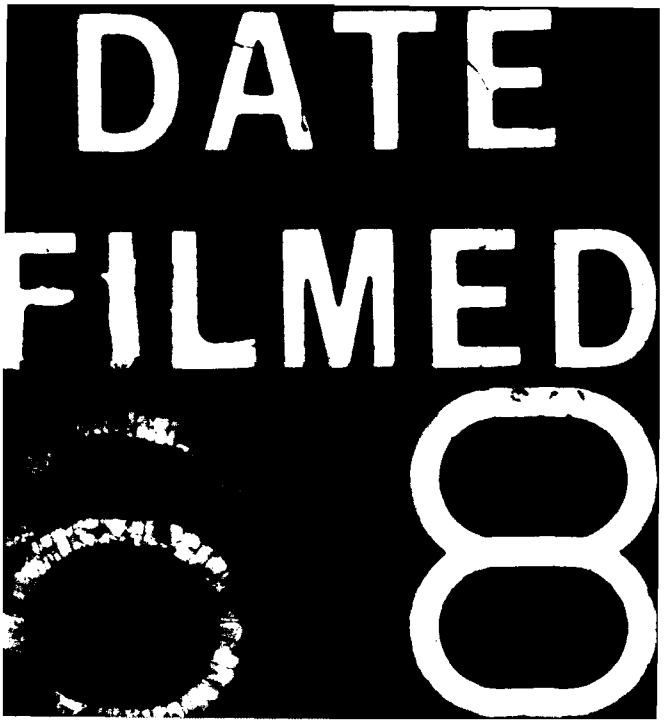
15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This Report presents criteria to which a development environment should conform to be suitable for the production of secure software. It gives a rationale for the criteria and a suitable security policy model for software development.


DATE
FILMED
8